

Notes on Modern C++

(standard c++11, 14, 17, 20)

Gyubeom Edward Im*

July 6, 2024

Contents

| | | |
|----------|---------------------------------------|-----------|
| 1 | Introduction | 1 |
| 2 | Intermediate | 1 |
| 2.1 | Temporary | 1 |
| 2.2 | Trivial constructor | 1 |
| 2.2.1 | Trivial default constructor | 1 |
| 2.2.2 | Trivial copy constructor | 1 |
| 2.3 | Type deduction | 2 |
| 2.4 | Auto type deduction | 3 |
| 2.5 | Array Name | 4 |
| 2.6 | Lvalue vs Rvalue | 5 |
| 2.7 | Reference & Overloading | 6 |
| 2.8 | Reference collapsing | 7 |
| 2.9 | Forwarding reference | 8 |
| 2.10 | Move constructor | 9 |
| 2.11 | std::move | 10 |
| 2.12 | Move and noexcept | 10 |
| 2.13 | Default move constructor | 12 |
| 2.13.1 | Rule of 3/5/0 | 12 |
| 3 | References | 13 |
| 4 | Revision log | 13 |

1 Introduction

2 Intermediate

2.1 Temporary

2.2 Trivial constructor

2.2.1 Trivial default constructor

생성자가 trivial하다는 말은 컴파일러가 자동으로 생성해주면서 동시에 아무 일도 하지 않을 때를 말한다

2.2.2 Trivial copy constructor

복사 생성자가 trivial하다는 말은 멤버변수 값을 복사하는 것 이외에 아무 일도 하지 않을 때를 말한다. 복사 생성자가 trivial하다면 배열 전체를 memcpy와 memmove 등으로 복사하는 것이 빠르다! 복사 생성자가 trivial하지 않다면 배열의 모든 요소에 대해 하나씩 “복사 생성자”를 호출해서 생성자를 호출해야 한다

*blog: alida.tistory.com, email: criterion.im@gmail.com

```

1 struct Point {
2     int x=0;
3     int y=0;
4 };
5
6 template<class T>
7 void constexpr copy_type(T* dst, T* src, std::size_t sz) {
8     if(std::is_trivially_copy_constructible_v<T>) {
9         std::cout << "using memcpy" << std::endl;
10        memcpy(dst, src, sizeof(T)*sz);
11    }
12    else {
13        std::cout << "using copy ctor" << std::endl;
14        while(sz--) {
15            new(dst) T(*src);
16            --dst, --src;
17        }
18    }
19 }
20
21 int main(){
22     Point arr1[5];
23     Point arr2[5];
24     copy_type(arr1, arr2, 5);
25 }
```

위 코드에서 Point 클래스는 int x,y와 같이 간단한 멤버변수만 존재하므로 trivial copy constructor이다. 하지만 virtual void foo() 같이 가상함수를 사용하거나 string s;와 같이 복사하는 클래스를 사용하게 되면 trivial하지 않게 된다. 이런 경우에는 **placement new** 또는 **std::construct_at**을 사용해야 한다.

2.3 Type deduction

컴파일 타임에 타입이 결정되는 auto 키워드에 대해 살펴보자

```

1 int main(){
2     int n=10;
3     const int c =10;
4
5     auto a1 = n; // int a1=n;
6     auto a2 = c; // (1) const int a2 = c; --> no.
7             // (2) int a2 = c;      --> ok.
8 }
```

type deduction(타입 추론)이 발생하는 키워드는 다음과 같다: **template**, **auto**, **decltype**

```

1 #include <iostream>
2 template<class T> void foo(T arg){
3     std::cout << typeid(T).name() << std::endl;
4 }
5 int main(){
6     int n=10;
7     foo(n);           // T=int
8     foo<const int&>(n); // T=const int&. But typeid(T).name() keep printing output 'int'
9 }
```

typeid(T).name()은 타입 이름만 추론할 뿐 const, volatile, reference 정보가 출력되지 않는다.

1. 이럴 때는 의도적으로 에러를 발생시켜서 정확한 타입을 에러 메시지를 통해 알 수 있다.
2. 또는 **boost::type_index** 라이브러리의 **type_id_with_cvr<T>().pretty_name()**을 사용하면 됨
3. 컴파일러가 제공하는 매크로를 사용하면 된다.
 - **__FUNCTION__**: 함수의 이름만 보여주므로 타입 추론에는 사용하지 않음
 - **__PRETTY_FUNCTION__**: g++, clang에서는 함수 이름과 타입이 나옴

- `__FUNCSIG`: cl.exe 버전

```

1 std::cout << __FUNCTION__ << std::endl;
2 std::cout << __PRETTY_FUNCTION__ << std::endl; // g++, clang
3 std::cout << __FUNCSIG__ << std::endl;           // cl.exe

```

T가 값인 경우를 살펴보자

```

1 #include <iostream>
2 template<class T> void foo(T arg){
3     while(--arg>0) {}
4 }
5
6 int main(){
7     int n=10;
8     int& r = n;
9     const int c = 10;
10    const int& cr = c;
11    foo(n); // T=int
12    foo(r); // T=int& 일 것 같지만 T=int
13    foo(c); // T=const int 일 것 같지만 T=int
14    foo(cr); // T=const int& 일 것 같지만 T=int
15 }

```

T 인자를 값으로 받을 때는 복사본 객체가 만들어져서 “**const, volatile, reference**” 속성을 제거하고 값만 받는다. 헷갈리는 것 중 하나가 값으로 받을 때는 인자의 const 속성은 제거되고 “**인자가 가리키는 곳의 const 속성을 유지**”한다. 무슨 이야기인지 살펴보자

```

1 #include <iostream>
2 template<class T> void foo(T arg){
3     std::cout << __PRETTY_FUNCTION__ << std::endl;
4 }
5 int main(){
6     const char* const s = "hello";
7     foo(s); // 포인터의 const 속성은 제거되지만 가리키는 곳 "hello"의 const 속성은 유지됨!
8         // const char* arg = "hello"가 됨!!
9 }

```

T 인자를 참조로 받을 때는 다음과 같다.

```

1 #include <iostream>
2 template<class T> void foo(T& arg){
3     std::cout << __PRETTY_FUNCTION__ << std::endl;
4 }
5
6 int main(){
7     int n = 10;          // T=int.      arg=int&
8     int& r = n;         // T=const int. arg=const int& (const 속성을 유지!)
9     const int c = 10;   // T=int        arg=int&. (T에서 reference 속성을 제거!)
10    const int& cr = c; // T=const int  arg=const int& (const 속성을 유지!)
11 }

```

주의해야 할 점은 T의 타입과 arg의 타입은 다르다는 것이다 (T& arg 이기 때문!). 함수 인자의 “reference를 제거하고 T의 타입을 결정한다”. 인자가 가진 “const, volatile 속성을 유지한다.”. 마지막으로 T에 배열이 전달된 경우를 살펴보자

```

1 template<class T> void foo(T arg){
2     std::cout << __PRETTY_FUNCTION__ << std::endl;
3 }
4 template<class T> void goo(T& arg){
5     std::cout << __PRETTY_FUNCTION__ << std::endl;
6 }
7
8 int main(){
9     int x[3] = {1,2,3};

```

```

10   foo(x); // T=int* 타입으로 받는다
11   goo(x); // T=int[3] 타입으로 받는다. arg는 int()[3] 타입으로 받는다
12 }
```

T& arg로 배열을 받는 경우 $\text{int } (\&\text{arg})[3] = \text{x}$; 처럼 받는게 되어서 배열의 reference가 된다. 따라서 아래와 같이 goo()를 사용하면 에러가 발생한다.

```

1 template<class T> void foo(T arg){
2     std::cout << __PRETTY_FUNCTION__ << std::endl;
3 }
4 template<class T> void goo(T& arg){
5     std::cout << __PRETTY_FUNCTION__ << std::endl;
6 }
7
8 int main(){
9     foo("orange", "apple"); // ok
10    goo("orange", "apple"); // error
11 }
```

foo와 **goo** 둘 다 const char 형식으로 받지만 포인터는 개수에 제한이 없으므로 ok인 반면에 reference는 const char[7], const char[6]은 다른 reference이기 때문에 같은 T를 받는 상황에서 에러가 발생한다!
foo(const char[7], const char[6]), **goo**(const char[7], const char[6])

2.4 Auto type deduction

template은 함수 인자로 추론하는 반면 **auto**는 우변의 표현식으로 타입을 추론한다. template을 ‘T arg = 함수 인자’처럼 추론한다고 볼 수 있으므로 사실 ‘auto a = 표현식’과 동일한 형태로 추론한다!

```

1 int main(){
2     int n=10;
3     int& r = n;
4     const int c = 10;
5     const int& cr = c;
6
7     auto a1 = n; // auto=int
8     auto a2 = r; // auto=int
9     auto a3 = c; // auto=int
10    auto a4 = cr; // auto=int
11
12    auto& a5 = n; // auto=int. a5=int&
13    auto& a6 = r; // auto=int. a6=int&
14    auto& a7 = c; // auto=const int. a7=const int&
15    auto& a8 = cr; // auto=const int. a8=const int&
16
17    int x[3] = {1,2,3};
18    auto a = x; // auto=int*
19    auto& b = x; // auto=int[3]. b=int(&)[3]
20 }
```

위와 같아 template에서 본 규칙들이 그대로 적용! 아래와 같이 조금 더 까다로운 경우를 보자.

```

1 int main(){
2     auto a1 = 1;
3     auto a2 = {1};
4     auto a3{1};
5
6     std::cout << typeid(a1).name() << std::endl; // auto=int
7     std::cout << typeid(a2).name() << std::endl; // auto=initialized_list
8     std::cout << typeid(a3).name() << std::endl; // auto=int
9     std::vector<int> v1(10,0);
10    std::vector<bool> v2(10,false);
11
12    auto a4 = v1[0];
13    auto a5 = v2[0];
14 }
```

```

15     std::cout << typeid(a4).name() << std::endl; // auto=int
16     std::cout << typeid(a5).name() << std::endl; // auto=temporary proxy 객체
17 }
```

배열은 auto a= 1라고 하면 배열 타입으로 추론된다. bool 타입은 최적화 과정에서 specialization되어 있다. 따라서 bool은 [] 연산자가 bool로 변환 가능한 **temporary proxy**으로 추론한다!

2.5 Array Name

배열의 이름은 배열의 1번째 요소의 주소로 암시적 형변환 된다.

```

1 int main(){
2     int x[3] = {1,2,3};
3
4     int *p0[3] = &x;    // error. 연산자 우선 순위에 따라 p[3], *p 순으로 추론된다.
5     int (*p1)[3] = &x; // ok. (*p)로 감싸줘야 x[3] 배열에 대한 제대로된 포인터가 된다.
6     int *p2 = x;      // ok. &x[0]
7
8     printf("%p, %p\n", p1, p1+1); // 배열 자체를 가리키므로 +1을 하면 12바이트만큼 증가한다.
9     printf("%p, %p\n", p2, p2+1); // 배열의 첫번째 원소를 가리키므로 +1을 하면 4바이트만큼 증가한다.
10
11    (*p1)[0] = 10;
12    *p2 = 10;
13 }
```

위 코드에서 보면 p1, p2가 가리키는 주소는 동일하지만 포인터 타입이 다르므로 주소를 해석하는 방식이 다르다. 다음과 같이 배열을 함수 인자를 받는 경우에 대해 알아보자

```

1 void f1(int p[3]) {
2     printf("%d\n", sizeof(p)); // 마치 3개 배열을 입력으로 받는 듯 보이지만 int *p와 동일한 포인터를 받고
3         // 있다. 즉, 12가 아닌 8(64bit)이 나온다
4 }
5
5 int main(){
6     int x[3] = {1,2,3};
7     f1(x);
8 }
```

함수 인자로 배열을 받을 때는 int *p와 같이 포인터 타입으로 받거나 int p[]와 같이 배열 타입으로 받는다. 이 때 컴파일러에 의해 둘 다 **포인터**로 변환한다. (int *p).

int p[3]와 같이 배열의 크기도 지정해줄 수 있는데 이 또한 **포인터**로 컴파일러가 변환한다(int *p). 헷갈리기 쉬운 문법이니 주의한다. 마지막으로 다차원 배열의 포인터에 대해 알아보자

```

1 void foo(int (*p)[2]) {
2     p[0][0] = 100;
3 }
4
4 int main(){
5     int y[3][2] = {1,2,3,4,5,6};
6
7     int (*p3)[3][2] = &y; // 배열의 변수와 정확히 동일한 형태를 유지하고 (*p)로 감싸주면 배열 포인터가 된다.
8     int (*p4)[2] = y;   // 배열의 첫번째 원소는 2차원 배열이므로 (=1,2) int (*p4)[2]와 같이 선언해줘야 한다.
9
10    foo(y);
11 }
```

2.6 Lvalue vs Rvalue

- lvalue: 등호의 왼쪽에 올 수 있는 표현식
- rvalue: 등호의 왼쪽에 올 수 없는 표현식

이외에도 c++에서는 다음과 같은 추가적인 구분 방법이 존재한다.

```
1 x=10;
```

```

2 int f1() { return x; }
3 int& f2() { return x; }
4
5 int main(){
6     int v1=0, v2=0;
7
8     v1=10; // ok. v1 : lvalue
9     10=v1; // error. 10 : rvalue
10    v2=v1;
11    int *p1 = &v1; // ok
12    int *p2 = &10; // error
13
14    f1() = 20; // error
15    f2() = 20; // ok
16    const int c = 10; // 상수도 lvalue의 특징을 가지고 있다 (이름, 주소)
17    c = 20; // error
18    "aa"[0] = 'x'; // error. lvalue 문제가 아니라
19                      // const char[3]이므로 에러 발생!
20}

```

이름, 주소가 있으면 lvalue, 없으면 rvalue. 참조를 반환하면 lvalue, 값을 반환하면 rvalue라고 볼 수 있다. 다음과 같은 의문이 들 수 있다.

1. **모든 상수는 rvalue인가?** : 상수는 immutable lvalue로 취급한다.

2. **모든 rvalue는 상수인가?** : Point(1,2).set(10,20) 같이 temporary 객체도 멤버 변수를 호출할 수 있으므로 상수가 아니다

lvalue, rvalue에 대한 혼란 오해 중 하나가 객체, 변수에 부여되는 속성으로 오해한다. **하지만 이는 표현식(expression)에 부여되는 속성이다!** 표현식이란 “하나의 값”을 만들어내는 코드 집합을 말한다!

```

1 int main(){
2     int n=3;
3
4     n=10; // ok
5     n+2 = 10; // error. n+2=5인데 이는 값이므로 rvalue!
6     n+2*3 = 10; // error.
7
8     (n=20) = 10; // ok. 표현식 ok
9
10    ++n = 10; // ok
11    n++ = 10; // error. n=3-->4로 바뀌는데 이는 값이므로 error
12}

```

어떤 값이 lvalue인지 rvalue인지 조사하려면 **decltype**을 사용하면 된다!

```

1 int main(){
2     int n = 10;
3
4     if(std::is_lvalue_reference_v<decltype(n++)>) // n++: rvalue, ++n: lvalue
5         std::cout << "lvalue" << std::endl;
6     else
7         std::cout << "rvalue" << std::endl;
8
9     if(std::is_lvalue_reference_v<decltype((n))>) // 그냥 n을 넣으면 rvalue로 잘못나온다. (n)을 통해
10        괄호로 감싸줘야 표현식으로 인식해서 lvalue로 정상 인식한다!
11        std::cout << "lvalue" << std::endl;
12    else
13        std::cout << "rvalue" << std::endl;
14}

```

다음과 같이 매크로를 만들어 놓으면 편하게 구분할 수 있다.

```

1 #define value_category(...)
2     if( std::is_lvalue_reference_v<decltype((__VA_ARGS__))> )
3         std::cout << "lvalue" << std::endl;
4     else if( std::is_rvalue_reference_v<decltype((__VA_ARGS__))> )

```

```

5     std::cout << "rvalue(xvalue" << std::endl;
6 else
7     std::cout << "rvalue(prvalue)" << std::endl;
8
9 int main(){
10    int n=10;
11
12    value_category(n);
13    value_category(n+2);
14    value_category(n++);
15    value_category(++n);
16 }
```

2.7 Reference & Overloading

참조자(reference)의 규칙에 대해 알아보자

```

1 int main(){
2    int n=3;
3
4    int& r1 = n; // ok
5    int& r2 = 3; // error
6    const int& r3 = n; // ok
7    const int& r4 = 3; // ok (하지만 상수성이 추가됨)
8
9    int&& r5 = n; // error
10   int&& r6 = 3; // ok. (상수 성질 없음! rvalue reference라고 부름)
11 }
```

임의의 숫자를 가리키고 싶을 땐 const int& r4 = 3과 같이 가리켜야 했으나 c++11 이후 rvalue-reference라는 문법이 등장하면서 int&& r6=3과 같이 가리킬 수 있게 되었음.

기존 int& r1을 **lvalue-reference**라고 부르며 int&& r6=3을 **rvalue-reference**라고 부름! 그렇다면 왜 상수성 없이 rvalue를 가리키는 것이 중요할까? 이는 move semantics와 perfect forwarding을 위해서 필요하다. 자세한 내용은 추후 다룰 예정이다. 다음으로 rvalue, lvalue의 함수 오버로딩에 대해 알아보자

```

1 class X{};
2
3 // void foo(X x) { std::cout << "X" << std::endl }. // 값 타입과 참조 타입은 서로 오버로딩될 수 없다!
4 void foo(X& x) { std::cout << "X&" << std::endl }           // 1
5 void foo(const X& x) { std::cout << "const X&" << std::endl }. // 2
6 void foo(X&& x) { std::cout << "X&&" << std::endl }         // 3
7 //void foo(const X&& x) { std::cout << "const X&&" << std::endl }
8
9 int main(){
10    X x;
11    foo(x); // lvalue. 어느 곳에 오버로딩 해야할 지 몰라서 에러 발생
12    // 값 타입을 주석처리하면 X x에 오버로딩 됨. (1, 2) 순서로 오버로딩
13    foo(X()); // rvalue. 어느 곳에 오버로딩 해야할지 몰라서 에러 발생
14    // 값 타입을 주석처리하면 X x에 오버로딩 됨. (3, 2) 순서로 오버로딩
15 }
```

foo 함수의 마지막 표기법 const X&& x는 문법적으로는 가능하지만 사용하지 않는다. Move semantic을 통해 대체할 수 있기 때문이다. 다음은 주로 헷갈리는 문법에 대해 알아보자.

```

1 class X{};
2
3 void foo(X& x) { std::cout << "X&" << std::endl }           // 1
4 void foo(const X& x) { std::cout << "const X&" << std::endl }. // 2
5 void foo(X&& x) { std::cout << "X&&" << std::endl }         // 3
6
7 int main(){
8    foo( X() );           // 3
9 }
```

```

10     X&& rx = X();
11     foo(rx);           // 3번이 호출될 것 같지만 1번이 호출됨! lvalue로 인식하기 때문
12     foo(static_cast<X&&>(rx)); // 3
13 }
```

X&& rx = X()에서 rx는 rvalue라고 생각할 수 있으나 **이름이 있으므로 lvalue로 취급된다.** 따라서 함수를 호출하면 1번이 호출된다!

따라서 3번을 호출하고 싶으면 static_cast<X&&>()을 사용하여 형변환을 하면 되는데 자세히 보면 **같은 타입을 왜 변환해야 하는가? 하는 의문이 생길 수 있다.** 이는 특수한 케이스로써 c++ 문법에서 타입 캐스팅이 아닌 value를 변환하는 캐스팅으로 기재되어 있다.

2.8 Reference collapsing

```

1 int main(){
2     int n=3;
3     int& lr = n; // lvalue reference
4     int&& rr = 3; // rvalue reference
5
6     int& &ref2ref = lr; // error! 명시적으로 레퍼런스를 가리키는 레퍼런스는 코딩 불가
7
8     decltype(lr)& r1 = ? // int& & ==> int&
9     decltype(lr)&& r2 = ? // int& && ==> int&
10    decltype(rr)& r3 = ? // int&& & ==> int&
11    decltype(rr)&& r4 = ? // int&& && ==> int&& 모두 두개씩 있을 때만 rvalue reference로 인식!
12 }
```

레퍼런스를 가리키는 레퍼런스는 명시적으로 코딩이 불가능하다. 하지만 decltype()을 통해 타입을 추론하게 되면 가능하다.

decltype(&&)&&인 경우에만 int&& 타입으로 추론하고 나머지 경우는 int&로 추론한다. 이런 추론 규칙을 **reference collapsing**이라고 한다! reference collapsing은 **typedef, using, decltype, template** 4가지 경우에 적용된다.

```

1 template<typename T> void foo(T&& arg) { }
2
3 int main(){
4     int n=3;
5
6     typedef int& lref;
7     lref&& r1 = n;      // int& && ==> int&
8
9     using rref = int&&;
10    rref&& r2 = 10;     // int&& && ==> int&&
11
12    decltype(r2)&& r3 = 10; // int&& && ==> int&&
13
14    foo<int&>(n);      // foo(int& && arg)
15                  // foo(int& arg) 함수 생성!
16 }
```

2.9 Forwarding reference

```

1 void f1(int& arg) {}
2 void f2(int&& arg) {}
3 template<typename T> void f3(T& arg) {}
4
5 int main(){
6     int n=3;
7
8     f1(n); // ok
9     f1(0); // error
```

```

10
11     f2(n); // error
12     f2(0); // ok
13
14     f3(n); // ok. T& - 128) * 64 + (' - 128) 꿀 어떤 값으로 받아도 lvalue reference이다.
15     f3(0); // error
16 }
```

T&는 어떤 타입으로 받아도 (int&, int&&) lvalue reference이다! 템플릿 케이스에 대해 보다 자세히 알아보자

```

1 template<typename T> void f3(T& arg) \{\}
2
3 int main()\{
4     int n=3;
5
6     // 사용자가 명시적으로 타입을 추론한 경우
7     f3<int>(n);
8     f3<int&>(n);
9     f3<int&&>(n); // 3가지 케이스 모두 int&로 추론되므로 lvalue만 올 수 있다!
10
11    f3(n); // ok. 사용자가 템플릿 인자를 전달하지 않으면 int로 추론한다
12    f3(0); // error
13 }
```

다음으로 템플릿 인자에 **T&&**가 붙어있는 경우를 생각해보자.

```

1 template<typename T> void f4(T&& arg) {}
2
3 int main()\{
4     int n=3;
5
6     f4<int>(n);      // int &&. ==> int&& rvalue reference
7     f4<int&>(n);    // int& && ==> int&. 이 케이스에서만 lvalue reference가 된다!
8     f4<int&&>(n);   // int&& && ==> int&& rvalue reference
9
10    f4(n); // ok. 컴파일러가 자동으로 int 타입으로 변환해서 넘겨준다 (int ==> int)
11    f4(0); // ok.
12 }
```

T&& 와 같이 작성하면 타입 추론 규칙에 따라 rvalue와 lvalue를 같이 전달할 수 있다! 헷갈리기 쉬운 것이 함수가 하나인데 두 인자를 받는것이 아니라 함수가 2개가 생성되어서 각각 따로 받는다. 그리고 생성된 각 함수는 call-by-value가 아닌 call-by-reference를 사용해서 전달받는다 이러한 T&& reference를 **forwarding(universal) reference**라고 부른다!

2.10 Move constructor

다음과 같이 얕은 복사(shallow copy)가 일어나는 경우를 살펴보자

```

1 class Person{
2     char* name;
3     int age;
4     public:
5     Person(const char*s, int a) : age(a) {
6         name = new char[strlen(s)+1];
7         strcpy_s(name, strlen(s)+1, s);
8     }
9     ~Person() { delete[] name; }
10 }
11
12 int main(){
13     Person p1("john", 20);
14     Person p2 = p1; // 얕은 복사 발생!
15 }
```

복사 생성자를 명시적으로 정해주지 않으면 컴파일러가 모든 멤버 변수 함수를 얕은복사하게 된다.

```

1 Person(const Person& p) : age(p.age) {
2     name = new char[strlen(p.name)+1];
3     strcpy_s(name, strlen(p.name)+1, p.name);
4 }
```

클래스 내에 다음과 같은 복사 생성자를 정의해주면 더 이상 얇은 복사가 발생하지 않고 깊은 복사가 발생한다! 하지만 복사 생성자는 성능 이슈가 존재한다. 다음과 같이 임시 객체를 반환하는 함수를 살펴보자

```

1 class Person{
2     char* name;
3     int age;
4     public:
5     Person(const char*s, int a) : age(a) {
6         name = new char[strlen(s)+1];
7         strcpy_s(name, strlen(s)+1, s);
8     }
9     ~Person() { delete[] name; }
10    Person(const Person& p) : age(p.age) {
11        name = new char[strlen(p.name)+1];
12        strcpy_s(name, strlen(p.name)+1, p.name);
13    }
14 }
15
16 Person foo() {
17     Person p("john", 20);
18     return p;
19 }
20
21 int main(){
22     Person ret = foo(); // 임시객체 반환하여 ret에 복사 생성자를 호출하여 복사해주고 바로 파괴됨!
23 }
```

foo() 함수는 값을 반환하므로 임시객체가 생성되어 ret에 복사 생성자를 통해 깊은 복사를 일으키고 바로 파괴된다. foo()의 임시 객체를 파괴하지 않고 ret에 임시 객체의 주소를 그대로 가리킨 뒤 임시 객체의 메모리를 0으로 만드는 방법이 효율적이다!

```

1 Person(Person&& p) : name(p.name), age(p.age) {
2     p.name = nullptr;
3 }
```

Person 클래스 내부에 위와 같이 move 생성자를 만들면 rvalue만 받을 수 있고 임시 객체들은 해당 함수를 호출한다. 해당 코드가 rvalue를 처리하는 경우 기존의 복사 생성자보다 메모리 효율적이다!

2.11 std::move

```

1 class Object{
2     Object() = default;
3     Object(const Object& obj) { std::cout << "copy ctor" << std::endl; }
4     Object(Object&& obj) { std::cout << "move ctor" << std::endl; }
5 }
6
7 Object foo() {
8     Object obj;
9     return obj;
10 }
11
12 int main(){
13     Object obj1;
14     Object obj2 = obj1; // copy
15     Object obj3 = foo(); // move
16     Object obj4 = static_cast<Object&&>(obj1); // move
17     Object obj5 = std::move(obj2);           // move. 위 긴 변환문을 간단하게 move 함수를 호출하여 해결할 수
18     있다.
```

std::move 함수를 호출하면 **obj1, obj2 같은 lvalue도 rvalue로 취급하여 move 생성자를 호출할 수 있다!**
obj1, obj2를 코드 내에서 더 이상 사용하지 않을 때 복사 생성자를 호출하기 보다 move 생성자를 호출하여 보다 효율적으로 값을 전달할 수 있다.

2.12 Move and noexcept

```
1 class Object {
2     public:
3     Object() = default;
4     Object(const Object&) { std::cout << "copy"; }
5     Object(Object&&) { std::cout << "move"; }
6 };
7
8 int main() {
9     std::vector<Object> v(3);
10    std::cout << "-----"; // copy, copy, copy 발생
11    v.resize(5);           // move, move, move 발생
12    std::cout << "-----";
13 }
```

v(3)로 처음 3개의 Object 자원을 확보한 후 resize를 통해 5개의 자원을 다시 확보한다고 5개 메모리를 새로 할당하고 기존 3개의 자원은 “**복사(copy)**”되어 새로운 메모리에 오게된다. 이렇게 vector의 베퍼를 새롭게 할당한 경우 결국 기존 베퍼를 제거하게 되므로 “**복사(copy)**”보다 “**이동(move)**”가 효율적이다! 하지만 위 코드를 실행하면 “**복사(copy)**”가 수행된다. 컴파일러가 기본적으로 복사를 수행하는 이유는 만약 이동을 하다가 예외가 발생하면 vector를 resize 이전 상태로 되돌릴 수 없다는 단점이 존재하기 때문이다.

따라서 이동을 사용하고 싶다면 되도록 예외가 발생하지 않도록 구현하고 **noexcept** 키워드를 붙여서 예외가 없음을 컴파일러에게 알려야 한다

```
1 class Object {
2     public:
3     Object() = default;
4     Object(const Object&) { std::cout << "copy"; }
5     Object(Object&&) noexcept { std::cout << "move"; } // 예외가 없음을 컴파일러에게 알림!
6 };
7
8 int main() {
9     Object o1;
10    Object o2 = o1; // copy
11    Object o3 = std::move(o1); // move
12    Object o4 = std::move_if_noexcept(o2); // move
13
14    std::vector<Object> v(3);
15    std::cout << "-----";
16    v.resize(5); // move, move, move 발생!
17    std::cout << "-----";
18 }
```

std::move_if_noexcept를 사용하면 컴파일러가 함수의 noexcept 키워드 유무를 검사하고 만약 키워드가 있다면 move를 실행한다. std::move_if_noexcept는 type_traits 기술로 예외 가능성을 조사한 후 예외 가능성이 있으면 const T& 타입으로 변환하고 예외 가능성이 없다면 T&& 타입으로 변환해주는 함수이다!

```
1 template<typename T>
2 constexpr std::conditional_t<
3     !std::is_nothrow_move_constructible_v<T> &&
4     std::is_copy_constructible_v<T>, const T&, T&>
5     move_if_noexcept(T& x) noexcept {
6         return std::move(x)
7     }
```

다음과 같이 클래스가 여러 멤버 변수를 가지고 있는 경우를 살펴보자

```
1 template<typename T>
2 class Object{
```

```

3   int n;
4   std::string s;
5   T t;
6
7 public:
8   Object() = default;
9   Object(const Object& other) : n(other.n), s(other.s), t(other.t) {}
10  Object(Object&& other) noexcept
11    : n(other.n),
12    s(std::move(other.s)),
13    t(std::move(other.t))
14  {}
15 };

```

복사 생성자를 보면 int n은 예외가 없다. 그리고 std::string s은 예외가 없음을 보장한다. 하지만 **임의의 타입 T는 예외가 존재할 가능성이 존재한다.** 이럴 땐 다음과 같이 키워드를 입력하면 된다.

```

1   Object(Object&& other) noexcept( noexcept( t(std::move(other.t)) ))
2     : n(other.n),
3     s(std::move(other.s)),
4     t(std::move(other.t))
5   {}

```

c++에서 noexcept는 두 가지 의미가 존재한다. 1) noexcept operator, 2) noexcept specifier

1. noexcept operator:

- bool b = noexcept(expression): 어떤 표현식이 예외 가능성이 있는지 조사한다

2. noexcept specifier:

- f() noexcept, f() noexcept(true): 함수 f는 예외가 없다.
- f() noexcept(false): 함수 f는 예외가 존재한다

또 다른 방법으로는 다음과 같이 쓸 수 있다.

```

1   Object(Object&& other) noexcept( std::is_nothrow_move_constructible_v<T> )
2     : n(other.n),
3     s(std::move(other.s)),
4     t(std::move(other.t))
5   {}

```

2.13 Default move constructor

move 생성자를 기본적으로 컴파일러가 제공하는 경우에 대해 살펴보자

```

1 class Object{
2   std::string name;
3 public:
4   Object() = default;
5   Object(const Object& other) : name(obj.name) {}           // [1] 복사 생성자
6   Object& operator=(const Object& obj) { name = obj.name; } // [2] 복사 대입 연산자
7   Object(Object&& obj) : name(std::move(obj.name)) {}      // [3] move 생성자
8   Object& operator=(Object&& obj) { name = std::move(obj.name); } // [4] move 대입 연산자
9 };

```

- case 1.** 사용자가 [1,2,3,4] 모두 제공하지 않는 경우 컴파일러가 [1,2,3,4] 대한 default 버전을 제공한다.
- case 2.** 사용자가 [1] (또는 [2])만 제공하는 경우 컴파일러는 [2](또는 [1])는 default 버전을 제공하지만 [3,4]는 제공하지 않는다. 사실 [1]만 제공하면 컴파일러가 [2]도 제공하지 않는 것이 맞지만 설계 상 오류로 [2]는 default 버전이 제공된다. (since c++98)
- case 3.** 사용자가 [3](또는 [4])를 제공하는 경우 컴파일러는 [1], [2]를 삭제해버린다. 즉, 사용할 수 없다. 그리고 [4](또는 [3])는 제공하지 않는다.

하지만 코딩을 하다보면 복사 생성자는 사용자가 제공하지만 move 계열 함수만 컴파일러에게 요청하고 싶다.
이런 경우에는 어떻게 해야할까?

```
1 class Object{
2     std::string name;
3     public:
4     Object() = default;
5     Object(const Object& other) = default;      // 복사 대입 연산자도 move 생성자를 default로 요청하면
6         반드시 같이 요청해야 한다.
7     Object(Object&& obj) = default;
8     Object& operator=(Object&& obj) = default; // default 버전을 요청한다!
9 };

```

(...)= default;로 default 버전을 요청하면 컴파일러가 move 생성자는 default 버전을 생성한다. 복사 대입 연산자도 move 생성자를 default로 요청할 때 같이 요청해야 한다! 위와 같은 코드 형태는 잘 작성된 오픈소스에서 많이 볼 수 있다!

2.13.1 Rule of 3/5/0

다음으로 널리 통용되는 규칙인 Rule of 3/5/0에 대해 알아보자

```
1 class Person {
2     char* name;
3     int age;
4     public:
5     Person(const char*s, int a) :age(a) {
6         name = new char[strlen(s)+1];
7         strcpy(name, strlen(s)+1, s);
8     }
9     // char*와 같이 포인터 멤버변수가 있고 동적으로 메모리를 할당한다면
10    // c++98 시절에는 소멸자/복사 생성자/복사 대입연산자를 반드시 만들어야 했다 (Rule of 3) !
11    // c++11 시절에는 소멸자/복사 생성자/복사 대입연산자/ move 생성자/move 대입연산자 또한 만들어야 한다.
12        (Rule of 5)
13 };

```

클래스 내부 변수에 포인터 멤버변수가 있고 동적으로 메모리를 할당한다면 반드시 선언해야 하는 함수를 가르켜 **Rule of 3 (c++98)**, **Rule of 5 (c++11)**라고 하였다.

char* 대신 std::string을 사용하면 사용자가 직접 자원을 관리할 필요가 없다. (= **Rule of 0**). 즉, STL에서 제공하는 클래스를 사용하면 클래스가 동적 메모리 할당에 대하여 컴파일러가 알아서 자동으로 제공한다. 결론은 STL을 많이 쓰면 좋다는 얘기이다.

2.14 Perfect forwarding

```
1 void foo(int n) {}
2 void goo(int& r) {r = 20; }

3

4 template<class F, class T>
5 void chronometry(F f, T arg) {
6     f(arg);
7 }

8

9 int main() {
10     int n = 10;
11     chronometry(foo, 10);
12     chronometry(goo, n);
13     std::cout << n << std::endl;
14 }
```

함수의 수행 시간을 측정해주는 chronometry 템플릿 함수를 정의해보자. 이를 통해 foo, goo를 둘 다 한 함수에서 처리하고 싶다. 이를 위해서는 perfect forwarding 기술이 필요하다. **perfect forwarding**이란 전달 받은 인자를 다른 함수에게 “값, const 속성, value category 등의 변화없이 그대로 전달”하는 것을 말한다

3 References

[1] (lecture) CODENURI - C++ Master

4 Revision log

- 1st: 2024-07-16